HW2-CSCI544

October 5, 2021

1 Importing Libraries

[14]:
```
import pandas as pd
import numpy as np
import nltk
import re
! pip install bs4
from bs4 import BeautifulSoup
import contractions
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize
import warnings
warnings.filterwarnings('ignore')
from sklearn.linear_model import Perceptron
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import Perceptron
from sklearn.metrics import accuracy_score, f1_score, recall_score,␣
↪precision_score

#from sklearn.svm import SVC


from sklearn.svm import LinearSVC as SVC
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import MultinomialNB

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader

from numpy import argmax
from copy import deepcopy
from numpy import vstack
```

```
! pip install gensim
import gensim
import gensim.downloader as api

import gensim.models
from gensim import utils
```

2 Dataset Generation

2.0.1 1. Dataset Generation (5 points)

2.0.2 We will use the Amazon reviews dataset used in HW1. Load the dataset and build a balanced dataset of 250K reviews along with their ratings (50K instances per each rating score) through random selection. Create ternary labels using the ratings. We assume that ratings more than 3 denote positive 1 sentiment (class 1) and rating less than 3 denote negative sentiment (class 2). Reviews with rating 3 are considered to have neutral sentiment (class 3). You can store your dataset after generation and reuse it to reduce the computational load. For your experiments consider a 80%/20% training/testing split.

2.0.3 Ans: Loading the dataset in the cell below, by first reading the tsv file and then removing any row of data having inconsistent value, I also drop all the columns of the dataset except the reviw_body and the star rating, since these are the only coulmns of use to us.

[2]:
```
#importing dataset
file_path = 'amazon_reviews_us_Kitchen_v1_00.tsv'
input_data = pd.read_csv(file_path,␣
 ↪sep='\t',error_bad_lines=False,warn_bad_lines=False)
input_data =input_data.dropna()

input_data = input_data[['review_body','star_rating']]
```

2.0.4 Ans: Created a balanced dataset in below cell by taking 50000 reviews for each rating score randomly and merging them together.

[3]:
```
#creating ternary reviews
review_1_data = input_data.loc[ input_data['star_rating'] == 1.0 ]
review_1_data = review_1_data.sample(n=50000, random_state=65)
review_2_data = input_data.loc[ input_data['star_rating'] == 2.0 ]
review_2_data = review_2_data.sample(n=50000, random_state=65)
review_3_data = input_data.loc[ input_data['star_rating'] == 3.0 ]
review_3_data = review_3_data.sample(n=50000, random_state=65)
review_4_data = input_data.loc[ input_data['star_rating'] == 4.0 ]
review_4_data = review_4_data.sample(n=50000, random_state=65)
review_5_data = input_data.loc[ input_data['star_rating'] == 5.0 ]
review_5_data = review_5_data.sample(n=50000, random_state=65)
```

```
input_data = pd.
 ↪concat([review_1_data,review_2_data,review_3_data,review_4_data,review_5_data])
input_data = input_data.sample(frac = 1, random_state=65).reset_index(drop=True)
```

2.0.5 Ans: In the below cell, assigned each row of data a ternary label based on their star rating

[ ]:
```
#creating ternary labels
input_data['ternary_label'] = input_data['star_rating']


input_data.loc[input_data['star_rating'] > 3.0, 'ternary_label'] = 1
input_data.loc[input_data['star_rating'] < 3.0, 'ternary_label'] = 2
input_data.loc[input_data['star_rating'] == 3.0, 'ternary_label'] = 3
```

2.0.6 Ans: Randomly splitting the dataset in the cell below into training and testing dataset with 80%/20% split.

[4]:
```
#splitting the dataset into training and testing for ternary reviews
training_dataset = input_data.sample(frac = 0.8, random_state=65)
testing_dataset = input_data.drop(training_dataset.index)
training_dataset = training_dataset.reset_index(drop=True)
testing_dataset = testing_dataset.reset_index(drop=True)
```

2.1 Data Cleaning

2.1.1 Performing data cleaning below on training and testing dataset, similar to HW1

[5]:
```
#Converting reviews in lower case
training_dataset['review_body'] = training_dataset['review_body'].str.lower()
testing_dataset['review_body'] = testing_dataset['review_body'].str.lower()

#removing HTML tags
training_dataset['review_body'] = training_dataset['review_body'].apply(lambda␣
 ↪x: BeautifulSoup(x, 'html.parser').get_text())
testing_dataset['review_body'] = testing_dataset['review_body'].apply(lambda x:␣
 ↪BeautifulSoup(x, 'html.parser').get_text())

#removing URL tags
training_dataset['review_body'] = training_dataset['review_body'].apply(lambda␣
 ↪x: re.sub(r'^https?:\/\/.*[\r\n]*', '', x))
testing_dataset['review_body'] = testing_dataset['review_body'].apply(lambda x:␣
 ↪re.sub(r'^https?:\/\/.*[\r\n]*', '', x))

#removing non-alphabetical characters
```

```
training_dataset['review_body'] = training_dataset['review_body'].str.
 ↪replace('[^a-zA-Z ]', ' ')
testing_dataset['review_body'] = testing_dataset['review_body'].str.
 ↪replace('[^a-zA-Z ]', ' ')

#removing the extra spaces between words
training_dataset['review_body'] = training_dataset['review_body'].apply(lambda␣
 ↪x: re.sub(' +', ' ', x))
testing_dataset['review_body'] = testing_dataset['review_body'].apply(lambda x:␣
 ↪re.sub(' +', ' ', x))

#performing contractions
def Contractionfunction(text):
    expanded_words = []
    for word in text.split():
        expanded_words.append(contractions.fix(word))

    expanded_text = ' '.join(expanded_words)
    return expanded_text

training_dataset['review_body'] = training_dataset['review_body'].apply(lambda␣
 ↪x: Contractionfunction(x))
testing_dataset['review_body'] = testing_dataset['review_body'].apply(lambda x:␣
 ↪Contractionfunction(x))
```

2.2 Pre-Processing

2.2.1 Performing Pre-Processing below on training and testing dataset, similar to HW1

[6]:
```
#removing stop words
StopWords = stopwords.words('english')

training_dataset['review_body'] = training_dataset['review_body'].apply(lambda␣
 ↪x: ' '.join([word for word in x.split() if word not in (StopWords)]))
testing_dataset['review_body'] = testing_dataset['review_body'].apply(lambda x:␣
 ↪' '.join([word for word in x.split() if word not in (StopWords)]))

#performing lemmatization
lemmatizer = WordNetLemmatizer()

def lemmatize_text(text):
    lemmatize_tokens = [lemmatizer.lemmatize(w) for w in word_tokenize(text)]
    return ' '.join(lemmatize_tokens)

training_dataset['review_body'] = training_dataset['review_body'].apply(lambda␣
 ↪x: lemmatize_text(x))
```

```
testing_dataset['review_body'] = testing_dataset['review_body'].apply(lambda x:␣
↪lemmatize_text(x))
```

3 Word Embedding

3.0.1 2. Word Embedding (30 points)

3.0.2 In this part the of the assignment, you will learn how to generate two sets of Word2Vec features for the dataset you generated. You can use Gensim library for this purpose. A helpful tutorial is available in the following link:

https://radimrehurek.com/gensim/auto_examples/tutorials/run_word2vec.html ### (a) (10 points) ### Load the pretrained "word2vec-google-news-300" Word2Vec model and learn how to extract word embeddings for your dataset. Try to check semantic similarities of the generated vectors using two examples of your own, e.g., King − Man + Woman = Queen or excellent outstanding. ### (b) (20 points) ### Train a Word2Vec model using your own dataset. Set the embedding size to be 300 and the window size to be 11. You can also consider a minimum word count of 10. Check the semantic similarities for the same two examples in part (a). What do you conclude from comparing vectors generated by yourself and the pretrained model? Which of the Word2Vec models seems to encode semantic similarities between words better?

3.1 Pretrained model

3.1.1 Part (a) Ans: In the cell below we load the pretrained model

[7]:
```
#loading the pretrained word2vec model
word_vec_pretrain_model = api.load('word2vec-google-news-300')
#print(word_vec_pretrain_model['Dinner'])
```

3.1.2 Part (a) Ans: In the cell below we check semantic similarities of the pretrained model on example:

3.1.3 1) Father - Boy + Mother = ?

3.1.4 2) Tea ~ Coffee

[69]:
```
print("For Pre-trained Model: Father - Boy + Mother = {}, with a cosine␣
↪similarity of {}".format(word_vec_pretrain_model.
↪most_similar(positive=["father", "mother"], negative=["boy"],␣
↪topn=1)[0][0],word_vec_pretrain_model.most_similar(positive=["father",␣
↪"mother"], negative=["boy"], topn=1)[0][1]))
print("For Pre-trained Model: Tea and Coffee have a cosine similarity of {}".
↪format(word_vec_pretrain_model.similarity("tea", "coffee")))
```

```
For Pre-trained Model: Father - Boy + Mother = husband, with a cosine similarity
of 0.7670101523399353
For Pre-trained Model: Tea and Coffee have a cosine similarity of
0.5635292530059814
```

3.2 Our model

3.2.1 Part (b) Ans: In the cell below we train a Word2Vec model using our own dataset and set the embedding size to be 300, the window size to be 11 and a minimum word count of 10. We also create a class MyCorpus to iteratively feed each review of the training dataset to the model.

[8]:
```
#train the Word2Vec model on our own dataset

class MyCorpus(object):

    def __iter__(self):

        for line in training_dataset['review_body'].tolist():
            yield utils.simple_preprocess(line)

sentences = MyCorpus()
model = gensim.models.
 ↪Word2Vec(sentences=sentences,vector_size=300,window=11,min_count=10)
```

3.2.2 Part (b) Ans: In the cell below we check semantic similarities of our model on example:

3.2.3 1) Father - Boy + Mother = ?

3.2.4 2) Tea ~ Coffee

[70]:
```
print("For Our Model: Father - Boy + Mother = {}, with a cosine similarity of␣
 ↪{}".format(model.wv.most_similar(positive=["father", "mother"],␣
 ↪negative=["boy"], topn=1)[0][0],model.wv.most_similar(positive=["father",␣
 ↪"mother"], negative=["boy"], topn=1)[0][1]))
print("For Our Model: Tea and Coffee have a cosine similarity of {}".
 ↪format(model.wv.similarity("tea", "coffee")))
```

```
For Our Model: Father - Boy + Mother = sister, with a cosine similarity of
0.6096693873405457
For Our Model: Tea and Coffee have a cosine similarity of 0.5299463868141174
```

3.2.5 Part (b) Ans: From the above results based on cosine similarity for 2nd example pretrained model encodes semantic similarity better, but the results for the 1st example indicate, intuitively our model gave the more logical result although with a lower cosine similarity, so overall no one model seems to be clearly better than the other.

3.2.6 In the cell below we create a binary training and testing dataset for class 1 and 2, (denoted by '_simplified'), we also write 2 functions to generate the word2vec feature vectors for each review in the dataset, one for our model and the other for pretrained model (genVec and genVec_pretrained respectively), these functions return the input(reviews) in the various format, that they will be required, to train the upcoming models(i.e, average word2vec vector, first 10 word2vec vector, first 50 word2vec vector representations for each review). We then store the output of these functions in appropriate variables to use in the code further below to train and test our various models.

[9]:
```
#creating binary dataset
training_dataset_simplified = training_dataset.loc[␣
 ↪training_dataset['star_rating'] != 3.0 ]
testing_dataset_simplified = testing_dataset.loc[␣
 ↪testing_dataset['star_rating'] != 3.0 ]
training_dataset_simplified = training_dataset_simplified.reset_index(drop=True)
testing_dataset_simplified = testing_dataset_simplified.reset_index(drop=True)

#functions for converting input to various word2vec vector representations
def genVec(reviews,rating_class,model):
    X = []
    y = []
    X_10 = []
    y_10 = []
    X_50 = []
    y_50 = []
    for i in range(len(reviews)):
        try:
            wordVecs = [model.wv[w] for w in reviews[i].split() if w in model.
 ↪wv.index_to_key]
            if len(wordVecs) > 0:
                senVec = np.mean([ele for ele in wordVecs], axis=0).tolist()
                X.append(senVec)
                y.append(rating_class[i])
            if len(wordVecs) >= 10:
                X_10.append(wordVecs[:10])
                y_10.append(rating_class[i])
            if len(wordVecs) >= 50:
                X_50.append(wordVecs[:50])
                y_50.append(rating_class[i])
            elif len(wordVecs) > 0:
                X_50.append(wordVecs)
```

```
                y_50.append(rating_class[i])
        except:
            pass
    return X, y, X_10, y_10, X_50, y_50

def genVec_pretrained(reviews,rating_class,model):
    X = []
    y = []
    X_10 = []
    y_10 = []
    X_50 = []
    y_50 = []
    for i in range(len(reviews)):
        try:
            wordVecs = [model[w] for w in reviews[i].split() if w in model]
            if len(wordVecs) > 0:
                senVec = np.mean([ele for ele in wordVecs], axis=0).tolist()
                X.append(senVec)
                y.append(rating_class[i])
            if len(wordVecs) >= 10:
                X_10.append(wordVecs[:10])
                y_10.append(rating_class[i])
            if len(wordVecs) >= 50:
                X_50.append(wordVecs[:50])
                y_50.append(rating_class[i])
            elif len(wordVecs) > 0:

                X_50.append(wordVecs)
                y_50.append(rating_class[i])
        except:
            pass
    return X, y, X_10, y_10, X_50, y_50

#storing the various inputs for training and testing, that will be feed to the␣
 ↪upcoming models
X_train_binary, y_train_binary, X_train_binary_10, y_train_binary_10,␣
 ↪X_train_binary_50, y_train_binary_50 =␣
 ↪genVec(training_dataset_simplified['review_body'],␣
 ↪training_dataset_simplified['ternary_label'],model)
X_test_binary, y_test_binary, X_test_binary_10, y_test_binary_10,␣
 ↪X_test_binary_50, y_test_binary_50 =␣
 ↪genVec(testing_dataset_simplified['review_body'],␣
 ↪testing_dataset_simplified['ternary_label'],model)
```

```
X_train_binary_pre, y_train_binary_pre, X_train_binary_pre_10,␣
 ↪y_train_binary_pre_10, X_train_binary_pre_50, y_train_binary_pre_50 =␣
 ↪genVec_pretrained(training_dataset_simplified['review_body'],␣
 ↪training_dataset_simplified['ternary_label'],word_vec_pretrain_model)
X_test_binary_pre, y_test_binary_pre, X_test_binary_pre_10,␣
 ↪y_test_binary_pre_10, X_test_binary_pre_50, y_test_binary_pre_50 =␣
 ↪genVec_pretrained(testing_dataset_simplified['review_body'],␣
 ↪testing_dataset_simplified['ternary_label'],word_vec_pretrain_model)

X_train, y_train, X_train_10, y_train_10, X_train_50, y_train_50 =␣
 ↪genVec(training_dataset['review_body'],␣
 ↪training_dataset['ternary_label'],model)
X_test, y_test, X_test_10, y_test_10, X_test_50, y_test_50 =␣
 ↪genVec(testing_dataset['review_body'],␣
 ↪testing_dataset['ternary_label'],model)

X_train_pre, y_train_pre, X_train_pre_10, y_train_pre_10, X_train_pre_50,␣
 ↪y_train_pre_50 = genVec_pretrained(training_dataset['review_body'],␣
 ↪training_dataset['ternary_label'],word_vec_pretrain_model)
X_test_pre, y_test_pre, X_test_pre_10, y_test_pre_10, X_test_pre_50,␣
 ↪y_test_pre_50 = genVec_pretrained(testing_dataset['review_body'],␣
 ↪testing_dataset['ternary_label'],word_vec_pretrain_model)
```

4 Simple models

4.0.1 3. Simple models (20 points)

4.0.2 Using the Word2Vec features that you can generate using the two models you prepared in the Word Embedding section, train a perceptron and an SVM model similar to HW1 for class 1 and class 2 (binary models). For this purpose, you can just use the average Word2Vec vectors for each review as the input feature . To improve your performance, use the data cleaning and preprocessing steps of HW1 to include only important words from each review when you compute the average. Report your accuracy values on the testing split for these models for each feature type along with values you reported in your HW1 submission, i.e., for each of perceptron and SVM, you need to report three accuracy values for "word2vec-google-news-300", your own Word2Vec, and TF-IDF features. What do you conclude from comparing performances for the models trained using the three different feature types (TF-IDF, pretrained Word2Vec, your trained Word2Vec)?

4.0.3 The function below calculates the accuracy, precision, recall and fscore given the predicted and actual output list

[95]:
```
def calc_aprf(name,model,y,y_pred):
    print('For {} model, {} has accuracy: {}, precision: {}, recall: {},␣
↪f-score: {}'.format(model,name,accuracy_score(y, y_pred),precision_score(y,␣
↪y_pred),recall_score(y, y_pred),f1_score(y, y_pred)))
```

4.1 Perceptron

4.2 Our word2vec model

4.2.1 Ans: In the cell below we train the perceptron on the binary label dataset with average word2vec vector representation build from our model for each review and then test it on the test set and display the results using the function we created.

[96]:
```
#training perceptron on our models average word2vec vector
perc = Perceptron(max_iter = 75, eta0 = 0.005, random_state=65)
perc.fit(X_train_binary, y_train_binary)

ytest_ppn = perc.predict(X_test_binary)

calc_aprf("Perceptron","Our",y_test_binary, ytest_ppn)
```

```
For Our model, Perceptron has accuracy: 0.8092027768727601, precision:
0.777787737540337, recall: 0.8673097106302164, f-score: 0.8201129462914393
```

4.3 Pre-trained word2vec model

4.3.1 Ans: In the cell below we train the perceptron on the binary label dataset with average word2vec vector representation build from the pre-trained model for each review and then test it on the test set and display the results using the function we created.

[97]:
```
#training perceptron on pretrained models average word2vec vector
perc = Perceptron(max_iter = 75, eta0 = 0.005, random_state=65)
perc.fit(X_train_binary_pre, y_train_binary_pre)

ytest_ppn = perc.predict(X_test_binary_pre)

calc_aprf("Perceptron","Pre-trained",y_test_binary_pre, ytest_ppn)
```

```
For Pre-trained model, Perceptron has accuracy: 0.714518351496931, precision:
0.6459736655045188, recall: 0.9532467532467532, f-score: 0.770090591772088
```

4.4 TF-IDF Model

4.4.1 Ans: In the cell below I display the results I got on the test dataset when TF-IDF Model was trained using perceptron

[106]:
```
print('For TF-IDF model, Perceptron has accuracy: 0.8655, precision: 0.8439,␣
 ↪recall: 0.8964, f-score: 0.8694')
```

```
For TF-IDF model, Perceptron has accuracy: 0.8655, precision: 0.8439, recall:
0.8964, f-score: 0.8694
```

4.4.2 Ans: Based on accuracy values on the testing dataset,from perceptron training, it seems that TF-IDF model performs the best, followed by the model we trained and then the pre-trained model. Thus TF-IDF is a more robust input feature, followed by our models' average word2vec representation and lastly the pretrained models average word2vec representation.

4.5 SVM

4.6 Our word2vec Model

4.6.1 Ans: In the cell below we train the SVM on the binary label dataset with average word2vec vector representation build from our model for each review and then test it on the test set and display the results using the function we created.

[107]:
```
#training SVM on our models' average word2vec vector

svc = SVC( max_iter = 1000, random_state=65)
svc.fit(X_train_binary, y_train_binary)

ytest_svm = svc.predict(X_test_binary)
```

```
calc_aprf("SVM","Our",y_test_binary, ytest_svm)
```

```
For Our model, SVM has accuracy: 0.8573218716322899, precision:
0.8632027603003857, recall: 0.850217402169024, f-score: 0.8566608756955459
```

4.7 Pre-trained word2vec model

4.7.1 Ans: In the cell below we train the SVM on the binary label dataset with average word2vec vector representation build from the pre-trained model for each review and then test it on the test set and display the results using the function we created.

[108]:
```
#training SVM on pre-trained models' average word2vec vector

svc = SVC( max_iter = 1000, random_state=65)
svc.fit(X_train_binary_pre, y_train_binary_pre)

ytest_svm = svc.predict(X_test_binary_pre)

calc_aprf("SVM","Pre-trained",y_test_binary_pre, ytest_svm)
```

```
For Pre-trained model, SVM has accuracy: 0.8177126393586371, precision:
0.8312538989394884, recall: 0.7987012987012987, f-score: 0.8146525371917668
```

4.8 TF-IDF Model

4.8.1 Ans: In the cell below I display the results I got on the test dataset when TF-IDF Model was trained using SVM

[103]:
```
print('For TF-IDF model, SVM has accuracy: 0.8930, precision: 0.8914, recall: 0.
↪8946, f-score: 0.8930')
```

```
For TF-IDF model, SVM has accuracy: 0.8930, precision: 0.8914, recall: 0.8946,
f-score: 0.8930
```

4.8.2 Ans: Based on accuracy values on the testing dataset,from SVM training, it seems that TF-IDF model performs the best, followed by the model we trained and then the pre-trained model. Thus TF-IDF is a more robust input feature, followed by our models' average word2vec representation and lastly the pretrained models average word2vec representation

5 Feedforward Neural Network

5.0.1 4. Feedforward Neural Networks (25 points)

5.0.2 Using the features that you can generate using the models you prepared in the Word "Embedding section", train a feedforward multilayer perceptron network for sentiment analysis classification. Consider a network with two hidden layers, each with 50 and 10 nodes, respectively. You can use cross entropy loss and your own choice for other hyperparamters, e.g., nonlinearity, number of epochs, etc. Part of getting good results is to select good values for these hyperparamters.

5.0.3 You can also refer to the following tutorial to familiarize yourself:

https://www.kaggle.com/mishra1993/pytorch-multi-layer-perceptron-mnist ### Although the above tutorial is for image data but the concept of training an MLP is very similar to what we want to do. ### (a) To generate the input features, use the average Word2Vec vectors similar to the "Simple models" section and train the neural network. Train a network for binary classification using class 1 and class 2 and also a ternary model for the three classes. Report accuracy values on the testing split for your MLP model for each of the binary and ternary classification cases. ### (b) (To generate the input features, concatenate the first 10 Word2Vec vectors for each review as the input feature and train the neuralnetwork. Report the accuracy value on the testing split for your MLP model for each of the binary and ternary classification cases. ### What do you conclude by comparing accuracy values you obtain with those obtained in the "'Simple Models" section (note you can compare the accuracy values for binary classification).

5.0.4 In the cell below we first define the class MLP which initiatlizes the Multi-layer Perceptron for us, whos' input is average word2vec vector representation for each review. The init function of the class defines all the layers being used in the network and the forward function defines the structure of the MLP model. Similarly, the class MLP_vec initializes Multi-layer Perceptron for us, whos' input is first 10 word2vec vector representations for each review.

5.0.5 The TrainData class helps in iteratively feeding the data to the model while training. And, the TestData class helps in iteratively feeding the data to the model while testing.

[110]:
```
class MLP(nn.Module):
    def __init__(self, classification = "binary", vocab_size = 300):
        super(MLP, self).__init__()
        hidden_1 = 50
        hidden_2 = 10
        self.fc1 = nn.Linear(vocab_size, hidden_1)
        self.fc2 = nn.Linear(hidden_1, hidden_2)
        if classification == "binary":
```

```
            self.fc3 = nn.Linear(hidden_2, 3)
        else:
            # For multi-classification
            self.fc3 = nn.Linear(hidden_2, 4)

        self.sig = nn.Sigmoid()
        self.soft = nn.Softmax(dim = 1)

    def forward(self, x):
        x = x.view(-1, x.shape[1])
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

class MLP_vec(nn.Module):
    def __init__(self, classification = "binary", vocab_size = 300):
        super(MLP_vec, self).__init__()
        hidden_1 = 50
        hidden_2 = 10
        if classification == "binary":
            self.fc3 = nn.Linear(hidden_2, 3)
        else:
            # For multi-classification
            self.fc3 = nn.Linear(hidden_2, 4)
        self.prod = 10
        self.fc1 = nn.Linear(vocab_size * self.prod, hidden_1)
        self.fc2 = nn.Linear(hidden_1, hidden_2)
        self.sig = nn.Sigmoid()
        self.soft = nn.Softmax(dim = 1)


    def forward(self, x):
        x = x.view(-1, x.shape[1] * x.shape[2])
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

class trainData(Dataset):

    def __init__(self, X_data, y_data):
        self.X_data = X_data
        self.y_data = y_data

    def __getitem__(self, index):
        return self.X_data[index], self.y_data[index]
```

```
    def __len__ (self):
        return len(self.X_data)


class testData(Dataset):

    def __init__(self, X_data, Y_data):
        self.X_data = X_data
        self.Y_data = Y_data

    def __getitem__(self, index):
        return self.X_data[index], self.Y_data[index]

    def __len__ (self):
        return len(self.X_data)
```

5.1 Our word2vec Model (binary classification using average word2vec vectors)

5.1.1 Part (a) Ans: In the cell below, we first initialize the model then choose its loss function and optimizer, then call the iterative training and testing data feeder functions from above. Afterwards we begin the training of the MLP network for the no of epochs given. And finally we evaluate the model based on the model saved after last epoch. The input of this network is the average word2vec vector generated by our model and output is either of the binary label

[114]:
```
# Initializing MLP with average word2vec vector from our model as input for␣
 ↪binary classification

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
#device = torch.device('cuda')
mlp_model = MLP() # binary classification

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(mlp_model.parameters(), lr=0.01)


mlp_model = mlp_model.to(device)
criterion = criterion.to(device)

training_data = trainData(torch.FloatTensor(X_train_binary), torch.
 ↪LongTensor(y_train_binary))
testing_data = testData(torch.FloatTensor(X_test_binary), torch.
 ↪LongTensor(y_test_binary))
```

```
train_loader = DataLoader(dataset = training_data, batch_size=16, shuffle =␣
 ↪True)
test_loader = DataLoader(dataset = testing_data, batch_size=16)

#Training the Model
n_epochs = 10
for epoch in range(n_epochs):

    train_loss = 0.0

    mlp_model.train()
    for input_data, label in train_loader:

        optimizer.zero_grad()
        output = mlp_model(input_data.to(device))
        loss = criterion(output, label.to(device))
        loss.backward()
        optimizer.step()
        train_loss += loss.item() * input_data.size(1)

    train_loss = train_loss/len(train_loader.dataset)

    #print('Epoch: {} \tTraining Loss: {:.6f}'.format(epoch+1, train_loss))
    torch.save(mlp_model.state_dict(), 'mlp_model' + str(epoch + 1) + '.pt')



# Evaluating the Model
mlp_model.load_state_dict(torch.load('mlp_model' +str(n_epochs) + '.pt'))
mlp_model = mlp_model.to('cpu')

predictions, actual = list(), list()
for test_data, test_label in test_loader:

    pred = mlp_model(test_data.to('cpu'))
    pred = pred.detach().numpy()
    pred = argmax(pred, axis= 1)
    target = test_label.numpy()
    target = target.reshape((len(target), 1))
    pred = pred.reshape((len(pred)), 1)
    pred = pred.round()
    predictions.append(pred)
    actual.append(target)

predictions, actual = vstack(predictions), vstack(actual)
acc = accuracy_score(actual, predictions)
prec = precision_score(actual, predictions)
```

```
recall = recall_score(actual, predictions)
f1 = f1_score(actual, predictions)
print('Using Our models average word2vec as input, for binary classification,␣
 ↪MLP has a Accuracy: {}, Precision: {}, Recall: {}, F1-Score: {}'.
 ↪format(acc,prec,recall,f1))
```

```
Using Our models average word2vec as input, for binary classification, MLP has a
Accuracy: 0.8683241021528283, Precision: 0.8518624505174799, Recall:
0.8926483082612824, F1-Score: 0.8717786021085514
```

5.2 Pre-Trained word2vec Model (binary classification using average word2vec vectors)

5.2.1 Part (a) Ans: In the cell below, we first initialize the model then choose its loss function and optimizer, then call the iterative training and testing data feeder functions from above. Afterwards we begin the training of the MLP network for the no of epochs given. And finally we evaluate the model based on the model saved after last epoch. The input of this network is the average word2vec vector generated by the pre-trained model and output is either of the binary label.

[115]:
```
# Initializing MLP with average word2vec vector from pretrained model as input␣
 ↪for binary classification

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
#device = torch.device('cuda')
mlp_model = MLP() # binary classification


criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(mlp_model.parameters(), lr=0.01)

mlp_model = mlp_model.to(device)
criterion = criterion.to(device)

training_data = trainData(torch.FloatTensor(X_train_binary_pre), torch.
 ↪LongTensor(y_train_binary_pre))
testing_data = testData(torch.FloatTensor(X_test_binary_pre), torch.
 ↪LongTensor(y_test_binary_pre))

train_loader = DataLoader(dataset = training_data, batch_size=16, shuffle =␣
 ↪True)
test_loader = DataLoader(dataset = testing_data, batch_size=16)

#Training the Model
n_epochs = 10
for epoch in range(n_epochs):
```

```
    train_loss = 0.0

    mlp_model.train()
    for input_data, label in train_loader:

        optimizer.zero_grad()
        output = mlp_model(input_data.to(device))
        loss = criterion(output, label.to(device)) #y_batch.unsqueeze(1) (label.
↪unsqueeze(1)).to(device)
        loss.backward()
        optimizer.step()
        train_loss += loss.item() * input_data.size(1)

    train_loss = train_loss/len(train_loader.dataset)

    #print('Epoch: {} \tTraining Loss: {:.6f}'.format(epoch+1, train_loss))
    torch.save(mlp_model.state_dict(), 'mlp_model_pre' + str(epoch + 1) + '.pt')



# Evaluating the Model

mlp_model.load_state_dict(torch.load('mlp_model_pre' +str(n_epochs) + '.pt'))
mlp_model = mlp_model.to('cpu')

predictions, actual = list(), list()
for test_data, test_label in test_loader:

    pred = mlp_model(test_data.to('cpu'))
    pred = pred.detach().numpy()
    pred = argmax(pred, axis= 1)
    target = test_label.numpy()
    target = target.reshape((len(target), 1))
    pred = pred.reshape((len(pred)), 1)
    pred = pred.round()
    predictions.append(pred)
    actual.append(target)

predictions, actual = vstack(predictions), vstack(actual)
acc = accuracy_score(actual, predictions)
prec = precision_score(actual, predictions)
recall = recall_score(actual, predictions)
f1 = f1_score(actual, predictions)
print('Using Pre-trained models average word2vec as input, for binary␣
↪classification, MLP has a Accuracy: {}, Precision: {}, Recall: {}, F1-Score:␣
↪{}'.format(acc,prec,recall,f1))
```

Using Pre-trained models average word2vec as input, for binary classification, MLP has a Accuracy: 0.8337467117624953, Precision: 0.8569828230022405, Recall: 0.8024475524475524, F1-Score: 0.8288190682556879

5.3 Our word2vec Model (Ternary classification using average word2vec vectors)

5.3.1 Part (a) Ans: In the cell below, we first initialize the model then choose its loss function and optimizer, then call the iterative training and testing data feeder functions from above. Afterwards we begin the training of the MLP network for the no of epochs given. And finally we evaluate the model based on the model saved after last epoch. The input of this network is the average word2vec vector generated by our model and output is anyone of the ternary label.

[118]:
```
# Initializing MLP with average word2vec vector from our model as input for␣
↪ternary classification

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
mlp_model = MLP(classification = "multi-class")

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(mlp_model.parameters(), lr=0.01)

mlp_model = mlp_model.to(device)
criterion = criterion.to(device)

training_data_multi = trainData(torch.FloatTensor(X_train), torch.
↪LongTensor(y_train))
testing_data_multi = testData(torch.FloatTensor(X_test), torch.
↪LongTensor(y_test))

train_loader_multi = DataLoader(dataset = training_data_multi, batch_size=16,␣
↪shuffle = True)
test_loader_mutli = DataLoader(dataset = testing_data_multi, batch_size=16)

#Training the Model
n_epochs = 10
for epoch in range(n_epochs):

    train_loss = 0.0

    mlp_model.train()
    for input_data, label in train_loader:

        optimizer.zero_grad()
        output = mlp_model(input_data.to(device))
```

```
        loss = criterion(output, label.to(device)) #y_batch.unsqueeze(1) (label.
↪unsqueeze(1)).to(device)
        loss.backward()
        optimizer.step()
        train_loss += loss.item() * input_data.size(1)

    train_loss = train_loss/len(train_loader.dataset)

    #print('Epoch: {} \tTraining Loss: {:.6f}'.format(epoch+1, train_loss))
    torch.save(mlp_model.state_dict(), 'mlp_model_multi' + str(epoch + 1) + '.
↪pt')


# Evaluating the Model
mlp_model.load_state_dict(torch.load('mlp_model_multi' +str(n_epochs) + '.pt'))
mlp_model = mlp_model.to('cpu')

predictions, actual = list(), list()
for test_data, test_label in test_loader:

    pred = mlp_model(test_data.to('cpu'))
    pred = pred.detach().numpy()
    pred = argmax(pred, axis= 1)
    target = test_label.numpy()
    target = target.reshape((len(target), 1))
    pred = pred.reshape((len(pred)), 1)
    pred = pred.round()
    predictions.append(pred)
    actual.append(target)

predictions, actual = vstack(predictions), vstack(actual)
acc = accuracy_score(actual, predictions)
prec = precision_score(actual, predictions)
recall = recall_score(actual, predictions)
f1 = f1_score(actual, predictions)
print('Using Our models average word2vec as input, for ternary classification,␣
↪MLP has a Accuracy: {}, Precision: {}, Recall: {}, F1-Score: {}'.
↪format(acc,prec,recall,f1))
```

```
Using Our models average word2vec as input, for ternary classification, MLP has
a Accuracy: 0.8319679318551922, Precision: 0.8605318745599306, Recall:
0.7936063936063936, F1-Score: 0.8257152508900033
```

5.4 Pre-trained word2vec Model (Ternary classification using average word2vec vectors)

5.4.1 Part (a) Ans: In the cell below, we first initialize the model then choose its loss function and optimizer, then call the iterative training and testing data feeder functions from above. Afterwards we begin the training of the MLP network for the no of epochs given. And finally we evaluate the model based on the model saved after last epoch. The input of this network is the average word2vec vector generated by the pre-trained model and output is anyone of the ternary label.

[120]:
```
# Initializing MLP with average word2vec vector from pre-trained model as input␣
↪for ternary classification

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
mlp_model = MLP(classification = "multi-class")

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(mlp_model.parameters(), lr=0.01)

mlp_model = mlp_model.to(device)
criterion = criterion.to(device)

training_data_multi = trainData(torch.FloatTensor(X_train_pre), torch.
↪LongTensor(y_train_pre))
testing_data_multi = testData(torch.FloatTensor(X_test_pre), torch.
↪LongTensor(y_test_pre))

train_loader_multi = DataLoader(dataset = training_data_multi, batch_size=16,␣
↪shuffle = True)
test_loader_mutli = DataLoader(dataset = testing_data_multi, batch_size=16)

#Training the Model
n_epochs = 10
for epoch in range(n_epochs):

    train_loss = 0.0

    mlp_model.train()
    for input_data, label in train_loader:

        optimizer.zero_grad()
        output = mlp_model(input_data.to(device))
        loss = criterion(output, label.to(device)) #y_batch.unsqueeze(1) (label.
↪unsqueeze(1)).to(device)
        loss.backward()
        optimizer.step()
```

```
        train_loss += loss.item() * input_data.size(1)

    train_loss = train_loss/len(train_loader.dataset)

    #print('Epoch: {} \tTraining Loss: {:.6f}'.format(epoch+1, train_loss))
    torch.save(mlp_model.state_dict(), 'mlp_model_multi_pre' + str(epoch + 1) +␣
 ↪'.pt')


# Evaluating the Model

mlp_model.load_state_dict(torch.load('mlp_model_multi_pre' +str(n_epochs) + '.
 ↪pt'))
mlp_model = mlp_model.to('cpu')

predictions, actual = list(), list()
for test_data, test_label in test_loader:

    pred = mlp_model(test_data.to('cpu'))
    pred = pred.detach().numpy()
    pred = argmax(pred, axis= 1)
    target = test_label.numpy()
    target = target.reshape((len(target), 1))
    pred = pred.reshape((len(pred)), 1)
    pred = pred.round()
    predictions.append(pred)
    actual.append(target)

predictions, actual = vstack(predictions), vstack(actual)

acc = accuracy_score(actual, predictions)
prec = precision_score(actual, predictions)
recall = recall_score(actual, predictions)
f1 = f1_score(actual, predictions)
print('Using Pre-trained models average word2vec as input, for ternary␣
 ↪classification, MLP has a Accuracy: {}, Precision: {}, Recall: {}, F1-Score:␣
 ↪{}'.format(acc,prec,recall,f1))
```

```
Using Pre-trained models average word2vec as input, for ternary classification,
MLP has a Accuracy: 0.826456219466366, Precision: 0.8746208869814021, Recall:
0.7634365634365634, F1-Score: 0.8152553673823176
```

5.5 Our word2vec Model (binary classification using first 10 word2vec vectors)

5.5.1 Part (b) Ans: In the cell below, we first initialize the model then choose its loss function and optimizer, then call the iterative training and testing data feeder functions from above. Afterwards we begin the training of the MLP network for the no of epochs given. And finally we evaluate the model based on the model saved after last epoch. The input of this network is the first 10 word2vec vector generated for each review, by our model and output is either of the binary label

[121]:
```
# Initializing MLP, with first 10 word2vec vector for each review, from our␣
↪model as input for binary classification

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
#device = torch.device('cuda')
mlp_model = MLP_vec() # binary classification


criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(mlp_model.parameters(), lr=0.01)

mlp_model = mlp_model.to(device)
criterion = criterion.to(device)

training_data = trainData(torch.FloatTensor(X_train_binary_10), torch.
↪LongTensor(y_train_binary_10))
testing_data = testData(torch.FloatTensor(X_test_binary_10), torch.
↪LongTensor(y_test_binary_10))

train_loader = DataLoader(dataset = training_data, batch_size=16, shuffle =␣
↪True)
test_loader = DataLoader(dataset = testing_data, batch_size=16)

#Training the Model
n_epochs = 10
for epoch in range(n_epochs):

    train_loss = 0.0

    mlp_model.train()
    for input_data, label in train_loader:

        optimizer.zero_grad()
        output = mlp_model(input_data.to(device))
        loss = criterion(output, label.to(device)) #y_batch.unsqueeze(1) (label.
↪unsqueeze(1)).to(device)
        loss.backward()
        optimizer.step()
```

```
        train_loss += loss.item() * input_data.size(1)

    train_loss = train_loss/len(train_loader.dataset)

    #print('Epoch: {} \tTraining Loss: {:.6f}'.format(epoch+1, train_loss))
    torch.save(mlp_model.state_dict(), 'mlp_model_vec' + str(epoch + 1) + '.pt')



# Evaluating the Model

mlp_model.load_state_dict(torch.load('mlp_model_vec' +str(n_epochs) + '.pt'))
mlp_model = mlp_model.to('cpu')

predictions, actual = list(), list()
for test_data, test_label in test_loader:

    pred = mlp_model(test_data.to('cpu'))
    pred = pred.detach().numpy()
    pred = argmax(pred, axis= 1)
    target = test_label.numpy()
    target = target.reshape((len(target), 1))
    pred = pred.reshape((len(pred)), 1)
    pred = pred.round()
    predictions.append(pred)
    actual.append(target)

predictions, actual = vstack(predictions), vstack(actual)
acc = accuracy_score(actual, predictions)
prec = precision_score(actual, predictions)
recall = recall_score(actual, predictions)
f1 = f1_score(actual, predictions)
print('Using Our models first 10 word2vec vector for each review as input, for␣
 ↪binary classification, MLP has a Accuracy: {}, Precision: {}, Recall: {},␣
 ↪F1-Score: {}'.format(acc,prec,recall,f1))
```

```
Using Our models first 10 word2vec vector for each review as input, for binary
classification, MLP has a Accuracy: 0.7515258909234102, Precision:
0.7474842539636575, Recall: 0.7165660351169408, F1-Score: 0.7316986747927149
```

5.6 Pre-trained word2vec Model (binary classification using first 10 word2vec vectors)

5.6.1 Part (b) Ans: In the cell below, we first initialize the model then choose its loss function and optimizer, then call the iterative training and testing data feeder functions from above. Afterwards we begin the training of the MLP network for the no of epochs given. And finally we evaluate the model based on the model saved after last epoch. The input of this network is the first 10 word2vec vector generated for each review, by the pre-trained model and output is either of the binary label

[123]:
```
# Initializing MLP, with first 10 word2vec vector for each review, from␣
↪pre-trained model as input for binary classification

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
#device = torch.device('cuda')
mlp_model = MLP_vec() # binary classification

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(mlp_model.parameters(), lr=0.01)

mlp_model = mlp_model.to(device)
criterion = criterion.to(device)

training_data = trainData(torch.FloatTensor(X_train_binary_pre_10), torch.
↪LongTensor(y_train_binary_pre_10))
testing_data = testData(torch.FloatTensor(X_test_binary_pre_10), torch.
↪LongTensor(y_test_binary_pre_10))

train_loader = DataLoader(dataset = training_data, batch_size=16, shuffle =␣
↪True)
test_loader = DataLoader(dataset = testing_data, batch_size=16)

#Training the model
n_epochs = 10
for epoch in range(n_epochs):

    train_loss = 0.0

    mlp_model.train()
    for input_data, label in train_loader:

        optimizer.zero_grad()
        output = mlp_model(input_data.to(device))
        loss = criterion(output, label.to(device)) #y_batch.unsqueeze(1) (label.
↪unsqueeze(1)).to(device)
        loss.backward()
```

```
        optimizer.step()
        train_loss += loss.item() * input_data.size(1)

    train_loss = train_loss/len(train_loader.dataset)

    #print('Epoch: {} \tTraining Loss: {:.6f}'.format(epoch+1, train_loss))
    torch.save(mlp_model.state_dict(), 'mlp_model_vec_pre' + str(epoch + 1) + '.
 ↪pt')



# Evaluating the Model

mlp_model.load_state_dict(torch.load('mlp_model_vec_pre' +str(n_epochs) + '.
 ↪pt'))
mlp_model = mlp_model.to('cpu')

predictions, actual = list(), list()
for test_data, test_label in test_loader:

    pred = mlp_model(test_data.to('cpu'))
    pred = pred.detach().numpy()
    pred = argmax(pred, axis= 1)
    target = test_label.numpy()
    target = target.reshape((len(target), 1))
    pred = pred.reshape((len(pred)), 1)
    pred = pred.round()
    predictions.append(pred)
    actual.append(target)

predictions, actual = vstack(predictions), vstack(actual)
acc = accuracy_score(actual, predictions)
prec = precision_score(actual, predictions)
recall = recall_score(actual, predictions)
f1 = f1_score(actual, predictions)
print('Using Pre-trained models first 10 word2vec vector for each review as␣
 ↪input, for binary classification, MLP has a Accuracy: {}, Precision: {},␣
 ↪Recall: {}, F1-Score: {}'.format(acc,prec,recall,f1))
```

```
Using Pre-trained models first 10 word2vec vector for each review as input, for
binary classification, MLP has a Accuracy: 0.7365906338973067, Precision:
0.7573954983922829, Recall: 0.6519061786480316, F1-Score: 0.7007027851113672
```

5.7 Our word2vec Model (ternary classification using first 10 word2vec vectors)

5.7.1 Part (b) Ans: In the cell below, we first initialize the model then choose its loss function and optimizer, then call the iterative training and testing data feeder functions from above. Afterwards we begin the training of the MLP network for the no of epochs given. And finally we evaluate the model based on the model saved after last epoch. The input of this network is the first 10 word2vec vector generated for each review, by our model and output is anyone of the ternary label

[124]:
```
# Initializing MLP, with first 10 word2vec vector for each review, from our␣
 ↪model as input for ternary classification

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
mlp_model = MLP_vec(classification = "multi-class")

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(mlp_model.parameters(), lr=0.01)

mlp_model = mlp_model.to(device)
criterion = criterion.to(device)

training_data_multi = trainData(torch.FloatTensor(X_train_10), torch.
 ↪LongTensor(y_train_10))
testing_data_multi = testData(torch.FloatTensor(X_test_10), torch.
 ↪LongTensor(y_test_10))

train_loader_multi = DataLoader(dataset = training_data_multi, batch_size=16,␣
 ↪shuffle = True)
test_loader_mutli = DataLoader(dataset = testing_data_multi, batch_size=16)

#Training the model
n_epochs = 10
for epoch in range(n_epochs):

    train_loss = 0.0

    mlp_model.train()
    for input_data, label in train_loader:

        optimizer.zero_grad()
        output = mlp_model(input_data.to(device))
        loss = criterion(output, label.to(device)) #y_batch.unsqueeze(1) (label.
 ↪unsqueeze(1)).to(device)
        loss.backward()
        optimizer.step()
        train_loss += loss.item() * input_data.size(1)
```

```
    train_loss = train_loss/len(train_loader.dataset)

    #print('Epoch: {} \tTraining Loss: {:.6f}'.format(epoch+1, train_loss))
    torch.save(mlp_model.state_dict(), 'mlp_model_multi_vec' + str(epoch + 1) +␣
 ↪'.pt')


# Evaluating the Model
mlp_model.load_state_dict(torch.load('mlp_model_multi_vec' +str(n_epochs) + '.
 ↪pt'))
mlp_model = mlp_model.to('cpu')

predictions, actual = list(), list()
for test_data, test_label in test_loader:

    pred = mlp_model(test_data.to('cpu'))
    pred = pred.detach().numpy()
    pred = argmax(pred, axis= 1)
    target = test_label.numpy()
    target = target.reshape((len(target), 1))
    pred = pred.reshape((len(pred)), 1)
    pred = pred.round()
    predictions.append(pred)
    actual.append(target)

predictions, actual = vstack(predictions), vstack(actual)
acc = accuracy_score(actual, predictions)
prec = precision_score(actual, predictions)
recall = recall_score(actual, predictions)
f1 = f1_score(actual, predictions)
print('Using Our models first 10 word2vec vector for each review as input, for␣
 ↪ternary classification, MLP has a Accuracy: {}, Precision: {}, Recall: {},␣
 ↪F1-Score: {}'.format(acc,prec,recall,f1))
```

```
Using Our models first 10 word2vec vector for each review as input, for ternary
classification, MLP has a Accuracy: 0.7335144156821678, Precision:
0.696965913347484, Recall: 0.7724347886251989, F1-Score: 0.7327622985789767
```

5.8 Pre-trained word2vec Model (ternary classification using first 10 word2vec vectors)

5.8.1 Part (b) Ans: In the cell below, we first initialize the model then choose its loss function and optimizer, then call the iterative training and testing data feeder functions from above. Afterwards we begin the training of the MLP network for the no of epochs given. And finally we evaluate the model based on the model saved after last epoch. The input of this network is the first 10 word2vec vector generated for each review, by the pre-trained model and output is anyone of the ternary label

[127]:
```
# Initializing MLP, with first 10 word2vec vector for each review, from␣
 ↪pre-trained model as input for ternary classification

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
mlp_model = MLP_vec(classification = "multi-class")

criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(mlp_model.parameters(), lr=0.01)

mlp_model = mlp_model.to(device)
criterion = criterion.to(device)

training_data_multi = trainData(torch.FloatTensor(X_train_pre_10), torch.
 ↪LongTensor(y_train_pre_10))
testing_data_multi = testData(torch.FloatTensor(X_test_pre_10), torch.
 ↪LongTensor(y_test_pre_10))

train_loader_multi = DataLoader(dataset = training_data_multi, batch_size=16,␣
 ↪shuffle = True)
test_loader_mutli = DataLoader(dataset = testing_data_multi, batch_size=16)

#Training the model
n_epochs = 10
for epoch in range(n_epochs):

    train_loss = 0.0

    mlp_model.train()
    for input_data, label in train_loader:

        optimizer.zero_grad()
        output = mlp_model(input_data.to(device))
        loss = criterion(output, label.to(device)) #y_batch.unsqueeze(1) (label.
 ↪unsqueeze(1)).to(device)
        loss.backward()
        optimizer.step()
```

```
        train_loss += loss.item() * input_data.size(1)

    train_loss = train_loss/len(train_loader.dataset)

    #print('Epoch: {} \tTraining Loss: {:.6f}'.format(epoch+1, train_loss))
    torch.save(mlp_model.state_dict(), 'mlp_model_multi_vec_pre' + str(epoch +␣
 ↪1) + '.pt')


# Evaluating the Model
mlp_model.load_state_dict(torch.load('mlp_model_multi_vec_pre' +str(n_epochs) +␣
 ↪'.pt'))
mlp_model = mlp_model.to('cpu')

predictions, actual = list(), list()
for test_data, test_label in test_loader:

    pred = mlp_model(test_data.to('cpu'))
    pred = pred.detach().numpy()
    pred = argmax(pred, axis= 1)
    target = test_label.numpy()
    target = target.reshape((len(target), 1))
    pred = pred.reshape((len(pred)), 1)
    pred = pred.round()
    predictions.append(pred)
    actual.append(target)

predictions, actual = vstack(predictions), vstack(actual)
acc = accuracy_score(actual, predictions)
prec = precision_score(actual, predictions)
recall = recall_score(actual, predictions)
f1 = f1_score(actual, predictions)
print('Using Pre-trained models first 10 word2vec vector for each review as␣
 ↪input, for ternary classification, MLP has a Accuracy: {}, Precision: {},␣
 ↪Recall: {}, F1-Score: {}'.format(acc,prec,recall,f1))
```

```
Using Pre-trained models first 10 word2vec vector for each review as input, for
ternary classification, MLP has a Accuracy: 0.7347907189841935, Precision:
0.7669217186580342, Recall: 0.6310800525842386, F1-Score: 0.6924011235102103
```

5.8.2 Ans: By comparing the accuracy values obtained for binary classification using average word2vec vector representation for each review, its clear that Multilayer Perceptron performs better than SVM and perceptron for both cases, when input is generated by our word2vec model and when input is generated by pre-trained word2vec model.

5.9 Recurrent Neural Networks

5.9.1 5. Recurrent Neural Networks (20 points)

5.9.2 Using the features that you can generate using the models you prepared in the "Word Embedding" section, train a recurrent neural network (RNN) for sentiment analysis classification. You can refer to the following tutorial to familiarize yourself:

https://pytorch.org/tutorials/intermediate/char_rnn_classification_tutorial.html ### (a) Train a simple RNN for sentiment analysis. You can consider an RNN cell with the hidden state size of 50. To feed your data into our RNN, limit the maximum review length to 50 by truncating longer reviews and padding shorter reviews with a null value (0). Train the RNN network for binary classification using class 1 and class 2 and also a ternary model for the three classes. Report accuracy values on the testing split for your RNN model. ### (b) Repeat part (a) by considering a gated recurrent unit cell.

5.9.3 In the cell below we first define the class rnnModel which initiatlizes the RNN network for us, whos' input is the first 50 word2vec vector representation for each review (if a review has less than 50 representations then we use padding). The init function of the class defines all the layers being used in the network and the forward function defines the structure of the RNN model. Similarly, the class gruModel initializes the GRU network for us, whos' input is the first 50 word2vec vector representation for each review (if a review has less than 50 representations then we use padding).

[128]:
```
class rnnModel(nn.Module):
    def __init__(self, input_size, output_size, hidden_dim, n_layers,␣
↪model_type = "rnn"):
        super(rnnModel, self).__init__()

        # Defining some parameters
        self.hidden_dim = hidden_dim
        self.n_layers = n_layers
        self.model_type = model_type

        #Defining the layers
        # RNN Layer
        self.layer = nn.RNN(input_size, hidden_dim, n_layers, batch_first=True)

        # Fully connected layer
        self.fc = nn.Linear(2500, output_size)
```

```
    def forward(self, x):

        batch_size = x.size(0)

        # Initializing hidden state for first input using method defined below
        hidden = self.init_hidden(batch_size)

        # Passing in the input and hidden state into the model and obtaining␣
↪outputs
        out, hidden = self.layer(x, hidden)

        # Reshaping the outputs such that it can be fit into the fully␣
↪connected layer
        out = out.contiguous().view(-1, out.shape[1] * out.shape[2])
        out = self.fc(out)

        return out, hidden

    def init_hidden(self, batch_size):
        # This method generates the first hidden state of zeros which we'll use␣
↪in the forward pass
        # We'll send the tensor holding the hidden state to the device we␣
↪specified earlier as well
        hidden = torch.zeros(self.n_layers, batch_size, self.hidden_dim)#.cuda()
        return hidden

class gruModel(nn.Module):
    def __init__(self, input_size, output_size, hidden_dim, n_layers,␣
↪model_type = "gru"):
        super(gruModel, self).__init__()

        # Defining some parameters
        self.hidden_dim = hidden_dim
        self.n_layers = n_layers
        self.model_type = model_type

        #Defining the layers
        # GRU Layer
        self.layer = nn.GRU(input_size, hidden_dim, n_layers, batch_first=True)

        # Fully connected layer
        self.fc = nn.Linear(2500, output_size)

    def forward(self, x):

        batch_size = x.size(0)
```

```
        # Initializing hidden state for first input using method defined below
        hidden = self.init_hidden(batch_size)

        # Passing in the input and hidden state into the model and obtaining␣
↪outputs
        out, hidden = self.layer(x, hidden)

        # Reshaping the outputs such that it can be fit into the fully␣
↪connected layer
        #out = out.contiguous().view(-1, self.hidden_dim)
        out = out.contiguous().view(-1, out.shape[1] * out.shape[2])
        out = self.fc(out)

        return out, hidden

    def init_hidden(self, batch_size):
        # This method generates the first hidden state of zeros which we'll use␣
↪in the forward pass
        # We'll send the tensor holding the hidden state to the device we␣
↪specified earlier as well
        hidden = torch.zeros(self.n_layers, batch_size, self.hidden_dim)#.cuda()
        return hidden
```

5.9.4 In the cell below rnn_train function is used to train any given RNN or GRU network for the no of epochs provided with the input features and output features provided. Similarly, rnn_test function is used to test the RNN or GRU model provided, for the epoch no provided, on the input and output features provided. The my_collate function is used to train the rnn or gru model using small batches of data.

[143]:
```
def my_collate(batch):
    data = [item[0] for item in batch]
    target = [item[1] for item in batch]

    return data, target

#used for training rnn or gru network
def rnn_train(model, epoch, dataset_x, dataset_y, name):
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    rnn_train = RNN_Data(dataset_x, dataset_y)
    train_loader_mode = DataLoader(dataset = rnn_train, batch_size=8, shuffle =␣
↪True, collate_fn=my_collate, drop_last=True)

    criterion = nn.CrossEntropyLoss()
    criterion = criterion.to(device)
    optimizer = torch.optim.Adam(model.parameters(), lr=0.0001)
```

```
    for ep in range(1, epoch + 1):

        for input_data, label in train_loader_mode:
            optimizer.zero_grad()
            input_data = torch.stack(input_data)
            #print(input_data.shape)
            label = torch.stack(label)
            #print(label.shape)
            output, hidden = model(input_data.to(device))
            #print(output.shape)
            #output = torch.tensor(output, dtype=torch.long)
            label = torch.tensor(label, dtype=torch.long)
            loss = criterion(output,label)
            loss.backward()
            optimizer.step()

        #print('Epoch: {} \tTraining Loss: {:.6f}'.format(ep, loss.item()))
        torch.save(model.state_dict(), name + str(ep) + '.pt')

#used for testing rnn or gru network
def rnn_test(model, epoch, dataset_x, dataset_y,␣
 ↪name,model_name,w2v_name,classify):
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    rnn_test = RNN_Data(dataset_x, dataset_y)
    test_loader_mode = DataLoader(dataset = rnn_test, batch_size=8,␣
 ↪collate_fn=my_collate, drop_last=True)


    model.load_state_dict(torch.load(name +str(epoch) + '.pt'))
    model = model.to(device)

    predictions, actual = list(), list()
    for test_data, test_label in test_loader_mode:
        test_data = torch.stack(test_data)
        test_label = torch.stack(test_label)
        pred, hid = model(test_data.to('cpu'))
        pred = pred.to('cpu')
        pred = pred.detach().numpy()
        pred = argmax(pred, axis= 1)
        target = test_label.numpy()
        target = target.reshape((len(target), 1))
        pred = pred.reshape((len(pred)), 1)
        pred = pred.round()
        predictions.append(pred)
        actual.append(target)
```

```
    predictions, actual = vstack(predictions), vstack(actual)
    acc = accuracy_score(actual, predictions)
    print('Using {} models first 50 word2vec vector for each review as input,␣
↪for {} classification, {} has a Accuracy: {}'.
↪format(w2v_name,classify,model_name,acc))
```

5.9.5 The RNN_Data class defined in the cell below, helps iteratively feed the data to the RNN or GRU model while training or testing. Note: when any input data is called which has less then 50 word vectors the remaining word vectors are padded with zeroes in the *getitem* function.

[144]:
```
class RNN_Data(Dataset):

    def __init__(self, X_data, Y_data):

        self.X_data = X_data
        self.Y_data = Y_data

    def __len__(self):

        return len(self.X_data)

    def __getitem__(self, index):
        pad = np.zeros((50, 300), dtype = float)
        pad[-len(self.X_data[index]):] = np.array(self.X_data[index])#[:50]
        X = torch.FloatTensor(pad)
        Y = torch.tensor(self.Y_data[index])

        return X, Y
```

5.9.6 Our word2vec model (binary classification using first 50 word2vec vectors on RNN)

5.9.7 Part (a) Ans: In the cell below we first construct our RNN model by calling the rnnModel class. Then we use the rnn_train function to train that model. Afterwards we use the rnn_test function to test the model saved after the last epoch on the testing set. The input of this network is the first 50 word2vec vector generated for each review, by our model and output is anyone of the binary label

[145]:
```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
rnn = rnnModel(300, 3, 50, 1)
rnn = rnn.to(device)

rnn_train(rnn, 5, X_train_binary_50, y_train_binary_50, name =␣
↪"rnn_binary_model")
```

```
rnn_test(rnn, 5, X_test_binary_50, y_test_binary_50, name = "rnn_binary_model",␣
↪model_name = "RNN", w2v_name="Our", classify="Binary")
```

```
Using Our models first 50 word2vec vector for each review as input, for Binary
classification, RNN has a Accuracy: 0.8583316623220373
```

5.9.8 Pre-trained word2vec model (binary classification using first 50 word2vec vectors on RNN)

5.9.9 Part (a) Ans: In the cell below we first construct our RNN model by calling the rnnModel class. Then we use the rnn_train function to train that model. Afterwards we use the rnn_test function to test the model saved after the last epoch on the testing set. The input of this network is the first 50 word2vec vector generated for each review, by the pre-trained model and output is anyone of the binary label

[146]:
```
rnn = rnnModel(300, 3, 50, 1)
rnn = rnn.to(device)

rnn_train(rnn, 5, X_train_binary_pre_50, y_train_binary_pre_50, name =␣
↪"rnn_pre_binary_model")
rnn_test(rnn, 5, X_test_binary_pre_50, y_test_binary_pre_50, name =␣
↪"rnn_pre_binary_model", model_name = "RNN", w2v_name="Pre-trained",␣
↪classify="Binary")
```

```
Using Pre-trained models first 50 word2vec vector for each review as input, for
Binary classification, RNN has a Accuracy: 0.8421026257767088
```

5.9.10 Our word2vec model (ternary classification using first 50 word2vec vectors on RNN)

5.9.11 Part (a) Ans: In the cell below we first construct our RNN model by calling the rnnModel class. Then we use the rnn_train function to train that model. Afterwards we use the rnn_test function to test the model saved after the last epoch on the testing set. The input of this network is the first 50 word2vec vector generated for each review, by our model and output is anyone of the ternary label

[147]:
```
rnn = rnnModel(300, 4, 50, 1)
rnn = rnn.to(device)

rnn_train(rnn, 5, X_train_50, y_train_50, name = "rnn_multi_model")
rnn_test(rnn, 5, X_test_50, y_test_50, name = "rnn_multi_model",  model_name =␣
↪"RNN", w2v_name="Our", classify="Ternary")
```

```
Using Our models first 50 word2vec vector for each review as input, for Ternary
classification, RNN has a Accuracy: 0.6977247236019869
```

5.9.12 Pre-trained word2vec model (ternary classification using first 50 word2vec vectors on RNN)

5.9.13 Part (a) Ans: In the cell below we first construct our RNN model by calling the rnnModel class. Then we use the rnn_train function to train that model. Afterwards we use the rnn_test function to test the model saved after the last epoch on the testing set. The input of this network is the first 50 word2vec vector generated for each review, by the pre-trained model and output is anyone of the ternary label¶

[148]:
```
rnn = rnnModel(300, 4, 50, 1)
rnn = rnn.to(device)

rnn_train(rnn, 5, X_train_pre_50, y_train_pre_50, name = "rnn_pre_multi_model")
rnn_test(rnn, 5, X_test_pre_50, y_test_pre_50, name = "rnn_pre_multi_model",␣
↪model_name = "RNN", w2v_name="Pre-trained", classify="Ternary")
```

```
Using Pre-trained models first 50 word2vec vector for each review as input, for
Ternary classification, RNN has a Accuracy: 0.6671604740550929
```

5.9.14 Our word2vec model (binary classification using first 50 word2vec vectors on GRU)

5.9.15 Part (b) Ans: In the cell below we first construct our GRU model by calling the gruModel class. Then we use the rnn_train function to train that model. Afterwards we use the rnn_test function to test the model saved after the last epoch on the testing set. The input of this network is the first 50 word2vec vector generated for each review, by our model and output is anyone of the binary label

[149]:
```
gru_model = gruModel(300, 3, 50, 1, model_type="gru")
gru_model = gru_model.to(device)

rnn_train(gru_model, 5, X_train_binary_50, y_train_binary_50, name =␣
↪"gru_binary_model")
rnn_test(gru_model, 5, X_test_binary_50, y_test_binary_50, name =␣
↪"gru_binary_model", model_name = "GRU", w2v_name="Our", classify="Binary")
```

```
Using Our models first 50 word2vec vector for each review as input, for Binary
classification, GRU has a Accuracy: 0.8769801483858031
```

5.9.16 Pre-trained word2vec model (binary classification using first 50 word2vec vectors on GRU)

5.9.17 Part (b) Ans: In the cell below we first construct our GRU model by calling the gruModel class. Then we use the rnn_train function to train that model. Afterwards we use the rnn_test function to test the model saved after the last epoch on the testing set. The input of this network is the first 50 word2vec vector generated for each review, by the pre-trained model and output is anyone of the binary label

[151]:
```
gru_model = gruModel(300, 3, 50, 1, model_type="gru")
gru_model = gru_model.to(device)

rnn_train(gru_model, 5, X_train_binary_pre_50, y_train_binary_pre_50, name =␣
↪"gru_pre_binary_model")
rnn_test(gru_model, 5, X_test_binary_pre_50, y_test_binary_pre_50, name =␣
↪"gru_pre_binary_model", model_name = "GRU", w2v_name="Pre-trained",␣
↪classify="Binary")
```

```
Using Pre-trained models first 50 word2vec vector for each review as input, for
Binary classification, GRU has a Accuracy: 0.8452345159350572
```

5.9.18 Our word2vec model (ternary classification using first 50 word2vec vectors on GRU)

5.9.19 Part (b) Ans: In the cell below we first construct our GRU model by calling the gruModel class. Then we use the rnn_train function to train that model. Afterwards we use the rnn_test function to test the model saved after the last epoch on the testing set. The input of this network is the first 50 word2vec vector generated for each review, by our model and output is anyone of the ternary label

[153]:
```
gru_model = gruModel(300, 4, 50, 1, model_type="gru")
gru_model = gru_model.to(device)

rnn_train(gru_model, 5, X_train_50, y_train_50, name = "gru_multi_model")
rnn_test(gru_model, 5, X_test_50, y_test_50, name = "gru_multi_model",␣
↪model_name = "GRU", w2v_name="Our", classify="Ternary")
```

```
Using Our models first 50 word2vec vector for each review as input, for Ternary
classification, GRU has a Accuracy: 0.6985058484217272
```

5.9.20 Pre-trained word2vec model (ternary classification using first 50 word2vec vectors on GRU)

5.9.21 Part (b) Ans: In the cell below we first construct our GRU model by calling the gruModel class. Then we use the rnn_train function to train that model. Afterwards we use the rnn_test function to test the model saved after the last epoch on the testing set. The input of this network is the first 50 word2vec vector generated for each review, by the pre-trained model and output is anyone of the ternary label

[154]:
```
gru_model = gruModel(300, 4, 50, 1, model_type="gru")
gru_model = gru_model.to(device)

rnn_train(gru_model, 5, X_train_pre_50, y_train_pre_50, name =␣
↪"gru_pre_multi_model")
rnn_test(gru_model, 5, X_test_pre_50, y_test_pre_50, name =␣
↪"gru_pre_multi_model", model_name = "GRU", w2v_name="Pre-trained",␣
↪classify="Ternary")
```

```
Using Pre-trained models first 50 word2vec vector for each review as input, for
Ternary classification, GRU has a Accuracy: 0.6803331197950032
```

[ ]:
```

```